COMP202 HW3

PSEUDOCODE

```
public class BinarySearchTree
{
   Node root;
         static class Node
         {
                  // Your code here
   int item;
   Node left;
   Node right;

         };
   public BinarySearchTree()
   {

      this.root=null;
   }
   public void insert(int data)
   {
      // Your code here
      Node newNode = new Node();
      newNode.item = data;
      if(root==null){
      root=newNode;
      return;
      }
      Node temp=root;

      while(temp!=null){

        if(data==temp.item){
          return;
        }
        if(data<temp.item){
          if(temp.left==null){
            temp.left=newNode;
            return;
          }
          temp=temp.left;

        }
        if(data>temp.item){
          if(temp.right==null){
```

```
          temp.right=newNode;
          return;
        }
        temp=temp.right;

      }

    }
  }
  public int height(Node node){

   if(node==null){
    return -1;
   }
   else{
    return 1+Math.max(height(node.left),height(node.right));
   }

  }
public boolean checkBalanced(Node node,boolean a)
  {

   if(node==null){
    return true;
   }
   if(a==false){
    return false;
   }
   if(Math.abs(height(node.left)-height(node.right))>1){
    a=false;
   }
   if(checkBalanced(node.left,a)==false){
    return false;
   }
   if(checkBalanced(node.right,a)==false){
    return false;
   }
    return true;
  }
  public boolean isBalanced()
  {
   return checkBalanced(root, true);
  }

      public void remove(int item)
      {
            // Your code here
  Node temp=root;
  Node parent=root;
```

```
while(temp!=null){
      if(temp.item==item){
        break;
      }

      if(item<temp.item){
        parent=temp;
        temp=temp.left;

      }
      if(temp==null){
      return;
    }
      if(item>temp.item){
       parent=temp;
        temp=temp.right;

      }
    }
     if(temp==null){
        return;
     }

    if(temp.right==null&&temp.left==null){
      if(parent.left!=null&&parent.left.item==temp.item){
        parent.left=null;
        temp=null;
        return;
      }
      if(parent.right!=null&&parent.right.item==temp.item){
        parent.right=null;
        temp=null;

        return;
      }
    }
    Node temp2=temp;

   if(height(temp2)==1){
     temp2=temp2.left;
     temp.item=temp2.item;
     temp.left=null;
     temp2=null;
     return;
   }
    temp2=temp2.right;
    Node parent2=temp2;
    while(temp2.left!=null){
      parent2=temp2;
      temp2=temp2.left;
```

```
      }
      temp.item=temp2.item;
      if(temp2.right!=null){
        parent2.left=temp2.right;
      }
      else{
        parent2.left=null;
      }
      temp2=null;
         }

         public boolean compareTo(BinarySearchTree tree)
         {

                  return compareTrees(this.root, tree.root);

         }
 public boolean compareTrees(Node root1, Node root2){

   if(root1==null&&root2==null){
     return true;
   }
   if(root1==null&&root2!=null){
     return false;
   }
   if(root1!=null&&root2==null){
     return false;
   }

   if(compareTrees(root1.left,root2.left)==false ||
compareTrees(root1.right,root2.right)==false || root1.item!=root2.item){
      return false;
     }
     return true;

  }

         public void printInOrder(Node node)
         {
                  if (node != null)
                  {
                           printInOrder(node.left);
                           System.out.print(node.item + " ");
                           printInOrder(node.right);
                  }
   }
}
```

Time Complexity Analysis:

Insert:

Until we get the end of the tree, we traverse it with temp=temp.left and temp=temp.right statements in while loop, so time complexity would be O(height), in worst case it would be O(n), in best case(if the tree is balanced) it would be O(logn).

Height:

In this function, function calls itself recursively for the left subtree and the right subtree I want to calculate time complexitys for 2 opposite ends, in a situation like tree looks like just a line:

T(n)=T(n-1)+c T(n-1)=T(n-2)+c
T(n)=T(n-2)+2c
.
.
.
T(n)=T(0)+ n.c T(0) is a constant so the time complexity for that case would be O(n).

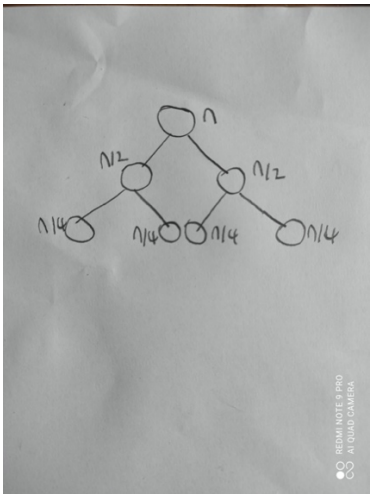
For a balanced tree, left subtree and right subtree would have same number of nodes, for that case:

T(n)=2T(n/2)+c T(n/2)=2T(n/4)+c
T(n)=4T(n/4) +3c T(n/4)=2T(n/8)+c
T(n)=8T(n/8) + 7c
.
.
.
T(n) = k.T(n/k) + k-1, to n/k be 1, k should be n so T(n) = n.O(1) + n-1 = O(n).

The complexity can also be found with a simple logic, you visit each node of tree and you simply add +1 except base case so it should be O(n).

CheckBalanced:

Function visits each node and calculate its height, with the boolean a parameter in my function, if one time height difference is more than one, function does not calculate height anymore but in the worst case (if tree is balanced), function visits each node and calculate its height. Height calculation is O(n) as I found above but there is a tricky part if my reasoning is correct, height calculation for above nodes are actually takes more time than belove, if the tree is like a complete balanced tree, height calculation for childs of a node takes half time of the parent so for a tree like this:



Time complexity would be n+2(n/2)+4(n/4). If we generalize that reasoning, it would be n+2(n/2)+4(n/4)+....k.(n/k) and k should be 2^height and height is logn. So our reasoning becomes n+n+n…+n and to find there is how many n's, we should look the equation like this: n+2^1.(n/2)+2^2.(n/2^2)…+2^logn.(n/2^logn). So there is logn n's so time complexity would be O(n.logn).

isBalanced:

it just calls checkBalanced so O(n.logn).

Remove:

Like the insert, we traverse the tree until we find the correct node to remove, if it's a leaf we just remove it, if its not we traverse the tree again to find smallest of the right subtree so anyway tree is been traversed until the end so time complexity would be O(height), in the worst case it would be O(n), in the best case it would be O(logn).

compareTrees:

In this function, both trees is being traversed in the same time(O(n+n)) and its elements are being compared(O(1)), so the time complexity for that function would be O(n).

compareTo:

This function just calls compareTrees so time complexity would be O(n).

Space Complexity Analysis

Insert:

A tree which has n nodes is created in the space in that function so space complexity would be O(n).

Height:

For each node one function is called in stack frame, for n nodes of the tree recursively n function is being called so space complexity is O(n).

checkBalanced:

For each node one function is called in stack frame, even though there is a height function called in each node, it will be returned until next function called. For n nodes of the tree recursively n function is being called so space complexity is O(n).

isBalance:

it calls checkBalanced so its O(n).

Remove:

Nothing is created and there is no recursive function so space complexity would be O(1).

compareTrees:

For each node one function is called in stack frame, for n nodes of the tree recursively n function is being called so space complexity is O(n).

compareTo:

it calls compareTrees so its O(n).

I have completed this assignment individually, without support from anyone else. I hereby accept that only the below listed sources are approved to be used during this assignment:

(i) Course textbook,

(ii) All material that is made available to me by the proffesor (e.g., via Blackboard for this course, course website, email from proffesor/TA),

(iii) Notes taken by me during lectures.

I have not used, accesed or taken any unpermitted information from any other source. Hence, all effort belongs to me.

Ali Taylan Akyürek